Computers Math. Applic. Vol. 24, No. 4, pp. 23–32, 1992
Printed in Great Britain.

0097-4943/92 $5.00 + 0.00

SINGULAR VALUE DECOMPOSITION ON SIMD HYPERCUBE AND SHUFFLE-EXCHANGE COMPUTERS

YI PAN
Department of Computer Science
University of Dayton, Dayton, OH 45469, U.S.A.
HENRY Y. H. CHUANG
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA 15260, U.S.A.

(Received February 1991)

Abstract—This paper reports several parallel singular value decomposition (SVD) algorithms on the hypercube and shuffle-exchange SIMD computers. Unlike previously published hypercube SVD algorithms which map a column pair of a matrix onto a processor, the algorithms presented in this paper map a matrix column pair onto a column of processors. In this way, a further reduction in time complexity is achieved. The paper also introduces the concept of two-dimensional shuffle-exchange networks, and corresponding SVD algorithms for one-dimensional and two-dimensional shuffle-exchange computers are developed.

1. INTRODUCTION

A singular value decomposition (SVD) of a real m-by-n ($m \geq n$) matrix A is its factorization into the product of three matrices:

$$A = UDV^T, \tag{1.1}$$

where U is an m-by-n matrix with orthogonal columns, D is an n-by-n non-negative diagonal matrix, and the n-by-n matrix V is orthogonal.[1] The n elements of D are called the singular values of matrix A. The singular value decomposition (SVD) technique has many applications [1], some of which require real-time computation. The SVD is perhaps the most important factorization of a real m-by-n ($m \geq n$) matrix.

The most common SVD method is the Golub-Kahan-Reinsch SVD algorithm [2,3] which requires $O(mn^2)$ time. In order to reduce the computation time, there has been much interest recently in developing faster SVD algorithms for various types of parallel computers and in designing algorithmically-specialized VLSI arrays such as the systolic arrays for SVD computation [3–10]. On a linear processor array, the most efficient SVD algorithm is the Jacobi-like algorithm given by Brent and Luk [4]. The algorithm, based on a one-sided orthogonalization method due to Hestenes [11], requires $O(n)$ processors and $O(mnS)$ time, where S is the number of sweeps.[2] Brent and Luk's algorithm is not optimal in terms of communication overhead. Unnecessary costs are incurred by mapping the systolic array architecture onto a ring-connected array, due to the double sends and receives required between pairs of neighboring processors. Eberlein [12], Bischof [13] and other have proposed various modifications for hypercube implementations, which require the embedding of rings via binary reflected *Gray Codes*. Gao and Thomas [14] have investigated this problem using a recursive divide-exchange communication pattern. An $O(nmS)$ time algorithm on a hypercube SIMD computer with $O(n)$ processors

The research of the first author was supported by an Andrew Mellon Predoctoral Fellowship awarded by the Mellon Foundation.

[1] We consider real matrices here, although the algorithm presented can be readily extended to handle complex matrices.

[2] $O(\log n)$ is the estimated number of sweeps required [4].

has been recently proposed by Chuang and Chen [10]. The algorithm, which is also based on Hestenes' method, has an improved computation time over that of Brent and Luk's on a linear processor array because of reduced data transfers among the processors.

In this paper, we present SVD algorithms on hypercube (cube connected) and shuffle-exchange architectures. Instead of mapping a column of data onto a processor in a hypercube, as is done in [10,14], we map a column pair of data onto a column of processors in a hypercube or a shuffle-exchange computer so that the total time is reduced.

This paper is organized as follows. The Hestenes method is reviewed in the next section. In Section 3, some notation and terms for hypercube and shuffle-exchange computers are introduced and two-dimensional shuffle-exchange computers are defined. In Section 4, we first show how the SVD can be carried out in $O(n \log m)$ time per sweep on a hypercube SIMD computer with $O(nm)$ processors. We then explain how the algorithm can be modified so that the number of processors is reduced to $O(nm/\log m)$ without increasing the time complexity. In Section 5, we describe the SVD algorithms on one-dimensional shuffle-exchange and two-dimensional shuffle-exchange computers. Section 6 deals with oversized SVD problems and Section 7 concludes the paper.

2. HESTENES METHOD

The basic idea of the decomposition is to generate an orthogonal matrix V such that the transformed matrix $AV = W$ has orthogonal columns. Normalizing the Euclidean length of each non-null column to unity, we get the relation

$$W = U'D.$$

Here, U' is a matrix whose non-null columns form an orthogonal set of vectors, and D is a non-negative diagonal matrix. An SVD of A is then given by

$$A = W V^T = U'D V^T. \tag{2.1}$$

As a null column of U' is always associated with a zero diagonal element of D, (1.1) and (2.1) are essentially identical.

Hestenes [11] uses plane rotations to construct V. A sequence of matrices $\{A_k\}$ is generated by

$$A_{k+1} = A_k R_k \qquad (k = 1, 2, 3, \ldots)$$

where $A_1 = A$, and each R_k is a plane rotation. Let $A_k = (a_1^k, a_2^k, \ldots, a_n^k)$, where a_i^k is the i-th column of A_k. Suppose $R_k = (r_{ij}^k)$ is a rotation on the (p, q) plane which orthogonalizes the columns a_p^k and a_q^k of A_k with $p < q$. R_k is an orthogonal matrix with all of its elements identical to those of the unit matrix, except that

$$\begin{aligned} r_{pp}^k &= \cos\theta, & r_{pq}^k &= \sin\theta, \\ r_{qp}^k &= -\sin\theta, & r_{qq}^k &= \cos\theta. \end{aligned}$$

We note that the post-multiplication of A_k by R_k affects only columns a_p^k and a_q^k, and that

$$(a_p^{k+1}, a_q^{k+1}) = (a_p^k, a_p^k). \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}. \tag{2.2}$$

The rotation angle θ should be chosen so that the two new columns are orthogonal. For this, we can use the formulas given by Rutishauser [15]. Defining:[3]

$$\gamma = (a_p^k)^T a_q^k, \quad \alpha = (a_p^k)^T a_p^k, \ \beta = (a_q^k)^T a_q^k, \tag{2.3}$$

[3] These are the inner products of the column vectors.

we set $\theta = 0$, if $\gamma = 0$, otherwise, we compute

$$L = \frac{\beta - \alpha}{2\gamma}, \quad t = \frac{\text{sign}(L)}{|L| + \sqrt{1 + L^2}},$$
$$\cos\theta = \frac{1}{\sqrt{1+t^2}}, \quad \sin\theta = t\cos\theta. \tag{2.4}$$

The rotation angle θ can always satisfy $|\theta| \leq \frac{\pi}{4}$. In this way, we orthogonalize the p-th and the q-th columns in the k-th step. The process in which all column pairs (i,j), for $i < j$, are orthogonalized exactly once is called a sweep. We repeat the sweep until A converges to W, which has orthogonal columns. In an actual computation, we can select a small positive number ε and stop the computation when the inner product of every column pair is less than ε. We use the ε as a threshold and orthogonalize a column pair only when its inner product is larger than ε. This threshold approach accelerates the convergence.

For a matrix with n columns, $n(n-1)/2$ column pairs have to be orthogonalized in a sweep. More than one column pairs can be orthogonalized simultaneously in a parallel system. In the orthogonalization, the elements of the new columns can be computed in parallel. The inner product operation to compute the rotation parameters γ, α, and β can also be carried out in parallel. The critical issue here is how to efficiently compute the rotation parameters, perform orthogonalization of column pairs and move columns to create all column pairs, using the available resources of the architecture.

3. PRELIMINARIES

This paper is concerned with the development of singular value decomposition for single instruction stream, multiple data stream (SIMD) parallel computers. All SIMD computers have the following characteristics:

(1) They consist of p processing elements (PEs). The PEs are indexed $0, 1, \ldots, p-1$, and an individual PE may be referenced as PE(i). Each PE is capable of performing the standard arithmetic and logic operations. In addition, each PE knows its index.
(2) Each PE has some local memory.
(3) The PEs are synchronized and operate under the control of a single instruction stream.
(4) Every processor receives data from the same neighbor at a given time. This implies that data can be transferred in a hypercube using only the connections of a particular dimension at a given time.
(5) An enable/disable mask can be used to select a subset of the PEs that are to perform an instruction. Only the enabled PEs will perform the instruction. The remaining PEs will be idle. All enabled PEs execute the same instruction. The set of enabled PEs can change from instruction to instruction.

A hypercube computer is defined as follows. Assume that $p = 2^q$ and let $i_{q-1}\ldots i_0$ be the binary representation of i, for $i \in [0, p-1]$. Let $i^{(b)}$ be the number whose binary representation is $i_{q-1}\ldots i_{b+1}\bar{i}_b i_{b-1}\ldots i_0$, where $0 \leq b < q$. In the hypercube model, PE(i) is connected to PI($i^{(b)}$), $0 \leq b < q$. Data can be transmitted from one PE to another PE only via the interconnection pattern.

The hypercube can also be logically looked upon as a two-dimensional structure, as shown in Fig. 1. Notice that the PEs are indexed in row-major order. That is, the PE at position (i,j) of the grid has index $iN + j$ in a hypercube of N PEs. For all our algorithms we will consider the two-dimensional equivalent of a hypercube with top-left PE indexed (0,0). A hypercube of size N can also be looked upon as two sub-hypercubes of size $N/2$. As an extension of this property, a hypercube of size $N = I_1 \times I_2 \times \cdots \times I_m$ can be deemed as m sub-hypercubes whose sizes are $I_1, I_2, \ldots, I_m$ respectively, where I_k, for $1 \leq k \leq m$, is a power of 2. Thus if PE(*,*) is denoted as a two-dimensional hypercube, a sub-hypercube with its second index equal to k is represented as PE($*, k$). These properties will be exploited in our algorithms.

Using the same symbols as above, we can define one-dimensional shuffle-exchange computers as follows. Define SHUFFLE(i) and UNSHUFFLE(i), respectively, to be the integers with binary

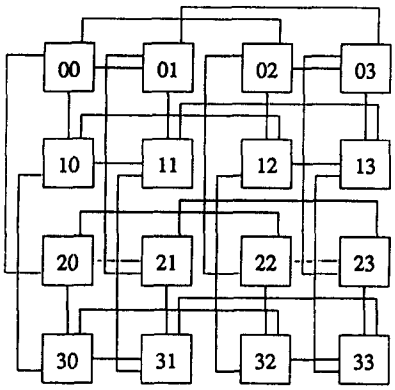

Fig. 1. A two-dimensional hypercube with 16 processors.

representations $i_{q-2}\, i_{q-3} \ldots i_0\, i_{q-1}$ and $i_0\, i_{q-1} \ldots i_1$. In the shuffle-exchange model, PE(i) is connected to PE($i^{(0)}$), PE(SHUFFLE(i)) and PE(UNSHUFFLE(i)). These three connections are called *exchange*, *shuffle* and *unshuffle*, respectively. Once again, data transmission from one PE to another is possible only via the connection scheme.

To extend the above definition, we can define two-dimensional shuffle-exchange computers as follows. Assume the two-dimensional shuffle-exchange computer is a $p \times q$ array and the PEs are indexed in row major order. In addition to the above three connections for a one-dimensional shuffle-exchange computer, there is another exchange connection; PE(i) is connected to PE($i^{\log q}$). In other words, in a two-dimensional shuffle-exchange computer of size $p \times q$, PE(i, j), where $0 \leq i < \log p$ and $0 \leq j < \log q$, is connected to PE(SHUFFLE(i), SHUFFLE(j)), PE(UNSHUFFLE(i), UNSHUFFLE(j)), PE(EXCHANGE(i), j), and PE(i, EXCHANGE(j)). Thus, in a two-dimensional shuffle-exchange computer, all rows and columns form a one-dimensional shuffle-exchange computer.

4. HYPERCUBE COMPUTER

In presenting our algorithms, we shall make use of conventions and notation similar to those used in [16]. We shall use brackets ([ ]) to index an array of memory locations and parentheses ('()') to index PEs. Thus, $B[i]$ refers to the i-th element of array B and $A(i)$ refers to the memory location A of PE(i). Then, $A[i](j)$ refers to the i-th element of array A in PE(j). The PEs are synchronized and operate under the control of a single instruction stream. The control unit broadcasts an instruction to all PEs, and all enabled PEs simultaneously execute the instruction. The enable/disable mask can be used to select a subset of the PEs that are to perform an instruction. For example, in the instruction

$$A(i) := A(i) + 1, \qquad (i_0 = 1),$$

$(i_0 = 1)$ is a mask that selects only those PEs whose binary index has bit 0 equal to 1; i.e., odd-indexed PEs increment their location A. Interprocessor assignments are denoted using the symbol $\leftarrow$, while intraprocessor assignments use the symbol $:=$. Thus, the assignment statement:

$$B(r^{(2)}) \leftarrow B(r), \qquad (r_2 = 0)$$

is executed only by PEs with bit 2 equal to 0. These PEs transmit the data in location B to location B of the PEs with bit 2 of address equal to 1.

In a *unit-route*, data may be transmitted from one PE to another if the two PEs are directly connected. We assume that links in the interconnection network are half-duplex. Hence, at any given time, data can be transferred only from one PE to another. The time complexity of an algorithm is the sum of the PE computation time and the communication time. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only these.

Procedure ORTH:

phase 1: $\gamma(i,j) = A(i,j) * B(i,j)$;
$\alpha(i,j) = A(i,j) * A(i,j)$;
$\beta(i,j) = B(i,j) * B(i,j)$;

phase 2: **for** $l := g$ **step** 1 **until** $f+g-1$ **do**
$\gamma(k) \leftarrow \gamma(k) + \gamma(k^{(l)})$
$\alpha(k) \leftarrow \alpha(k) + \alpha(k^{(l)})$
$\beta(k) \leftarrow \beta(k) + \beta(k^{(l)})$
end for l;

phase 3: **If** $\|\gamma(0,j)\| > \varepsilon$, **then**

$$L(0,j) = \frac{\beta(0,j) - \alpha(0,j)}{2\gamma(0,j)};$$

$$t(0,j) = \frac{\text{sign}[L(0,j)]}{\|L(0,j)\| + (1 + L(0,j)^2)^{1/2}};$$

$$c(0,j) = \frac{1}{(1 + (t(0,j))^2)^{1/2}};$$

$$s(0,j) = t(0,j) * c(0,j);$$

end If;

phase 4: **for** $l := g$ **step** 1 **until** $f+g-1$ **do**
$c(k^{(l)}) \leftarrow c(k)$, $(k_\ell = 0)$
$s(k^{(l)}) \leftarrow s(k)$, $(k_\ell = 0)$
end for l;

phase 5: $A(i,j) := A(i,j) * c(i,j) - B(i,j) * s(i,j)$;
$B(i,j) := A(i,j) * s(i,j) + B(i,j) * c(i,j)$;

end ORTH;

4.1 Hypercube with $m(n/2)$ Processors

Consider a hypercube SIMD computer with $m(n/2) = 2^{f+g}$ processors, where $f = \log m$ and $g = \log(n/2)$. For the purpose of the discussion, we shall map the processors in the hypercube onto an $m \times (n/2)$ rectangular array. Assume that the processors in the rectangular array are indexed in row major order; i.e., the processor in position (i,j), denoted as PE(i,j), has the index value $i(n/2) + j$. Note that the array indices range $[0, m-1]$ for rows, and $[0, (n/2)-1]$ for columns. If the mapping is such that the binary representation of the index of a processor in the rectangular array equals the binary address of the corresponding processor in the hypercube, then the processor in the hypercube with binary address $r_{f+g-1}, \ldots, r_0$ maps to PE(i,j), where $i = r_{f+g-1}, \ldots, r_g$ and $j = r_{g-1}, \ldots, r_0$. An f-dimensional subcube, therefore, maps onto a column of the array. Each PE(i,j) has several registers; registers $A(i,j)$ and $B(i,j)$ are used to store matrix elements. The given matrix A is initially stored as follows:

$$A(i,j) = a_{ij}, \quad B(i,j) = a_{i,\frac{n}{2}+j}, \qquad \text{for } 0 \le i < m, \quad 0 \le j < \frac{n}{2},$$

where a_{ij} is the (i,j) element of matrix A. Thus, the n columns of the matrix are partitioned into $n/2$ pairs and each column pair is stored in a column of processors.

A column of m processors orthogonalizes the column pair stored in it by using the procedure ORTH given below. The procedure ORTH is divided into several distinct phases. In the first, the products

$$\gamma(i,j) = A(i,j) * B(i,j), \qquad 0 \le i < m,$$
$$\alpha(i,j) = A(i,j) * A(i,j), \qquad 0 \le i < m,$$
$$\beta(i,j) = B(i,j) * B(i,j), \qquad 0 \le i < m$$

are calculated in the m processors. In the second phase, the sums $\sum_{i=0}^{m-1} \gamma(i,j)$, $\sum_{i=0}^{m-1} \alpha(i,j)$, and $\sum_{i=0}^{m-1} \beta(i,j)$ are computed by combining the partial sums using the cube connections, and the results are stored in PE$(0,j)$. This is accomplished by first adding the numbers stored in the two processors connected by the lowest dimension of the f-dimensional subcube. The partial

sums in the two processors connected by the second lowest dimension are then added, and so on. PE$(0,j)$ will have the final accumulated sums which are the inner products. In the third phase, the values $L(j)$, $t(j)$, $c(j)$ and $s(j)$ are computed locally in $P(0,j)$. In the fourth phase, the rotation parameters $c(0,j) = c(j)$ and $s(0,j) = s(j)$ are broadcast over the cube connections to all processors in the column. In the fifth phase, elements of the two new matrix columns are computed locally using the rotation parameters received.

After the procedure ORTH is performed in parallel on all $n/2$ columns of processors, half of the newly computed matrix columns are exchanged in parallel between neighboring columns of processors using the cube connections. The procedure ORTH is then carried out on all columns of processors in parallel again. In this way, the parallel execution of ORTH and the parallel exchange of matrix columns are repeated $(n-1)$ times to complete a sweep. The whole process is given in the procedure CCSVD for an m-by-n matrix on a system with $m \times (n/2)$ processors connected as an $(f+g)$-dimensional cube.

The exchanges of matrix columns can be done in parallel by using the connections of the cube which can be specified by cube functions:

$$\text{Cube}_k(r_{g+f-1} \ldots r_k \ldots r_1\ r_0) = r_{f+g-1} \ldots \overline{r_k} \ldots r_1\ r_0, \qquad (k = 0,1,2,\ldots,f+g-1).$$

Call the matrix columns stored in the A registers and the B registers, A-columns and B-columns, respectively. All different combinations of an A-column and a B-column can be obtained by moving the A-columns around to meet all B-columns. To do this, we apply the connection functions $\text{Cube}_{s(1)}$, $\text{Cube}_{s(2)}, \ldots,$ $\text{Cube}_{s(2^g-1)}$ in sequence. Here $s(j)$ is j-th number in the exchange sequence defined in [16], and is the position number of the rightmost 1 in the binary representation of j (lowest position is 0). For instance, when $g = 3$, $s(1)$ to $s(7)$ are 0,1,0,2,0,1,0. Therefore, by applying Cube_0, Cube_1, Cube_0, Cube_2, Cube_0, Cube_1, Cube_0 in that order, column A_j (for $j = 1,2,\ldots,7$) will visit all B-columns once (lines 4 to 8 of CCSVD). In order to obtain all possible column pairs, we still have to consider the pairing among the A-columns and among the B-columns. This can be achieved by iteratively applying the process of dividing a subcube in half, exchanging the A-columns in one half with the B-columns in the other, and then moving the A-columns around within the two subcubes (lines 9 to 12 of CCSVD). In procedure CCSVD, each iteration of the "**while**" body is a sweep and each iteration of the "**for** k" loop is a traversal of the A-columns within a $(g-k)$-dimensional subcube.

```
     Procedure CCSVD:
1    While not (all |γ| < ε) do
2      ORTH;
3      for k := 0 step 1 until g − 1 do
4       for p := 1 step 1 until 2^(g−k) − 1 do
5         h := S(p);
6         A(i,j) ← A(i,j^(h));
7         ORTH;
8       end for p;
9      if r_k = 0 then
10       A(i,j) ← B(i,j^(k)),  (r_k = 0);
11     else
12       B(i,j) ← A(i,j^(k)),  (r_k = 1);
13     ORTH;
14     end for k;
15   end While;
```

Now we analyze the total time complexity of algorithm CCSVD. In algorithm ORTH, phases 1, 3, and 5 take constant time. Both phases 2 and 4 run through $f = \log m$ steps, and so each takes $O(\log m)$. Therefore, the computing time for ORTH is $O(\log m)$.

The processing time of the algorithm CCSVD is dominated by its two loops. The number of times a statement in the inner loop is executed is:

$$\sum_{k=0}^{g-1}\left(\sum_{p=1}^{2^{g-k}-1} 1\right) = \left(\frac{n}{2}-1\right)+\left(\frac{n}{4}-1\right)+\left(\frac{n}{8}-1\right)+\cdots+1 \le n.$$

Within each inner loop, procedure ORTH is executed once and two matrix columns are exchanged in parallel, which takes 1 time unit for full duplex links (2 time units for half duplex links). Therefore, an iteration of the inner loop takes $O(\log m)$ time and a sweep takes $O(n \log m)$ time. Also, notice that the storage space used in each processor is constant. The efficiency of the algorithm is

$$e = \frac{t_s}{t_p \times p} = \frac{O(n^2 m)}{O(n \log m) \times (m)(n/2)} = \frac{1}{\log m},$$

where $t_s = O(n^2 m)$ is the computing time in a single processor system, and t_p is the computing time in a parallel system with p processors.

4.2. Hypercube with $(mn)/(2 \log m)$ Processors

A slightly modified form of the algorithm uses fewer processors to achieve the same time complexity. Consider an $(m/\log m) \times (n/2)$ rectangular array with cube connections. A column of $m/\log m$ processors contains a column pair of data, with the local memory of each processor storing $\log m$ pairs of data. Now, each processor takes $O(\log m)$ time to multiply the $\log m$ pairs. Adding these products locally takes $O(\log m)$ additional time. Adding the partial sums into PE$(0, j)$ and broadcasting the rotation parameters to the other processors in the same column costs $O(\log(m/\log m))$ time, since the number of processors in a column is $m/\log m$. Finally, computing the new column pair takes $O(\log m)$ time since $\log m$ pairs are processed by each processor. Therefore, the total time required by procedure ORTH is kept within $O(\log m)$. The only difference in procedure CCSVD, as a result of the change, is the data exchange between two columns of processors. Since each processor contains $\log m$ pairs of data now, the exchange of two matrix columns takes $O(\log m)$ time. Therefore, an iteration of the inner loop still takes $O(\log m)$ time, and the whole CCSVD algorithm takes $O(n \log m)$ also. However, the storage space required in each processor is $O(\log m)$ instead of constant. The efficiency of the modified algorithm is now constant because

$$e = \frac{t_s}{t_p \times p} = \frac{O(n^2 m)}{O(n \log m) \times (m/\log m)(n/2)} = \text{constant},$$

where $t_s = O(n^2 m)$ is the computing time in a single processor system, and t_p is the computing time in a parallel system with p processors. This represents a linear speedup in computing time.

5. SHUFFLE-EXCHANGE COMPUTERS

In this section, we discuss the SVD implementations for the shuffle-exchange computer (SEC). Consider a one-dimensional shuffle-exchange computer with $n/2$ PEs, such that every PE has a local memory large enough to store a pair of columns. Suppose that initially the two columns are stored in arrays $A[i](j)$ and $B[i](j)$, $0 \le i < m-1$, $0 \le j < n/2$, respectively. The algorithm is similar to CCSVD except that each data transfer now involves a vector of m elements instead of only one datum, and the data routing must use the connections of a shuffle-exchange computer. In a shuffle-exchange computer, we can transfer data between two nodes whose addresses differ only in the leftmost position. Therefore, we must simulate the data transmission of the hypercube. In the hypercube, PE(r) is directly connected to PE$(r^{(d)})$, and the basic data transfer is PE$(r) \leftarrow$ PE$(r^{(d)})$, for $0 \le d < P$. Clearly, the following subroutine for the shuffle-exchange computer implements the basic data transmisison of the hypercube.

Note that SHUFFLE and UNSHUFFLE are complementary operations. If several UNSHUFFLE operations are followed by the same number of SHUFFLE operations, the data will be

```
    Procedure TRANSMIT (R: register; d: integer)
1   for i := 0 to d do
2     R(UNSHUFFLE(k)) ← R(k)
3   end
4   R(EXCHANGE(k)) ← R(k)
5   for i := 0 to d do
6     R(SHUFFLE(k)) ← R(k)
7   end
```

transferred to their original positions. Therefore, replacing the data transfer $A(i) \leftarrow A(i^{(d)})$ by the procedure TRANSMIT(A, d), we obtain a one-dimensional shuffle-exchange algorithm for singular value decomposition (1DSESVD).

Now we analyze the time complexity of algorithm 1DSESVD. Obviously, lines 6, 7, 10 and 12 dominate the time complexity of the algorithm. The time spent in procedure ORTH is $O(m)$, since each PE needs to process a column pair whose length is m. The number of times line 7 is executed is:

$$\sum_{k=0}^{g-1}\left(\sum_{p=1}^{2^{g-k}-1} 1\right) = \left(\frac{n}{2}-1\right)+\left(\frac{n}{4}-1\right)+\left(\frac{n}{8}-1\right)+\cdots+1 \le n.$$

Therefore, the total time spent on line 7 is $O(mn)$.

According to the definition of an exchange sequence, it is easily shown that the number of zero's is at least half of the sequence length, the number of one's is at least a quarter of the sequence length, and so on. In general, in an exchange sequence of length P, the number of integer i's, $1 \le i < P-1$, is $2^{\log p - i - 1}$. For each integer i in X_p, there is a data transfer $A(k^{(i)}) \leftarrow A(k)$ in the above operations. Therefore, via procedure TRANSMIT, it will take $2i+1$ unit-routes in the shuffle-exchange computer. The total time for lines 4–8 for a particular k is

$$\sum_{i=0}^{2^{g-k}-1}(2i+1)*2^{g-k-i-1} < \sum_{i=1}^{\infty}(2(i-1)+1)*2^{g-k-i}$$

$$< 2*2^{g-k}\sum_{i=1}^{\infty}\frac{i}{2^i} - 2^{g-k}\sum_{i=1}^{\infty}\frac{1}{2^i} = 3*2^{g-k}.$$

Since k ranges from 0 to $g-1$ and each data transfer on line 6 involves a vector transfer (m elements), the total time spent on line 6 is calculated as follows:

$$m\sum_{k=0}^{g-1}3(2^{g-k}) = m\,3\left\{\left(\frac{n}{2}\right)+\left(\frac{n}{4}\right)+\left(\frac{n}{8}\right)+\cdots\right\} \le 3mn.$$

The time spent on line 10 is at most $O(m(\log n)^2)$, since simulating a step in a hypercube by the procedure TRANSMIT takes at most $\log n$ steps, line 10 is executed $\log n$ times, and each transfer involves a vector of m elements. Similarly, the time spent on line 12 is $O(m(\log n)^2)$. Therefore, the total time for algorithm 1DSESVD is $O(mn)$, which is on the same order of magnitude as that for a hypercube of the same size [10].

The algorithm 1DSESVD can be improved by mapping the array onto a two-dimensional shuffle-exchange computer with $m \times n/2$ PEs. Instead of mapping a column pair onto a processor, now we map it onto a column of processors of the two-dimsional shuffle-exchange computer. Thus, a column of m processors orthogonalizes the column pair stored in it by using the procedure ORTH1 described below. Procedure ORTH1 is divided into several distinct phases. In the first, the products

$$\gamma(i,j) = A(i,j)*B(i,j), \qquad 0 \le i < m,$$

$$\alpha(i,j) = A(i,j)*A(i,j), \qquad 0 \le i < m,$$

$$\beta(i,j) = B(i,j)*B(i,j), \qquad 0 \le i < m$$

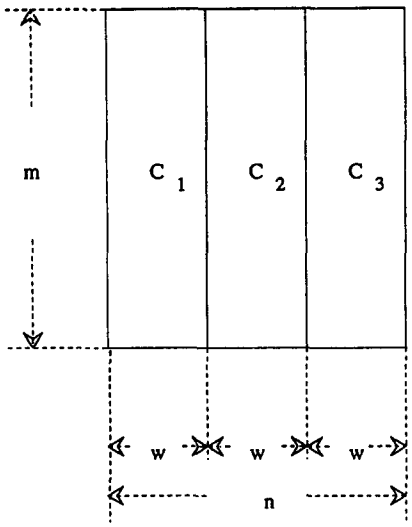

Fig. 2. Partition of a large matrix into column groups.

are calculated in the m processors. In the second phase, the sums $\sum_{i=0}^{m-1} \gamma(i,j)$, $\sum_{i=0}^{m-1} \alpha(i,j)$, and $\sum_{i=0}^{m-1} \beta(i,j)$ are computed by combining the partial sums, and the results are stored in $PE(0,j)$. In the third phase, the values $L(j)$, $t(j)$, $c(j)$ and $s(j)$ are computed locally in $P(0,j)$. In the fourth phase, the rotation parameters $c(0,j) = c(j)$ and $s(0,j) = s(j)$ are broadcast over the connections to all processors in the column. Finally, in the fifth phase, elements of the two new matrix columns are computed locally using the rotation parameters received.

In algorithm ORTH1, phases 1, 3 and 5 take constant time. Phase 2 requires $O(\log m)$ time, since summation of m data elements can be completed on a one-dimensional shuffle-exchange computer of m processors in $O(\log m)$ time [16]. Similarly, phase 4 takes $O(\log m)$ as broadcast can be performed on a one-dimensional shuffle-exchange computer in logarithmic time. Therefore, the computing time for ORTH1 is $O(\log m)$. The whole algorithm is the same as the one specified in CCSVD except that all data transfers on cube connections will be replaced by procedure TRANSMIT, and that ORTH is replaced by ORTH1. Hence, the time analysis remains the same as that for the one-dimensional shuffle-exchange algorithm except for lines 6, 7, 10, and 12. The time spent on line 6 is $O(n)$ because only one datum is transmitted each time. Similarly, lines 10 and 12 take $O\left((\log n)^2\right)$ time. Line 7 is executed n times as before, and procedure ORTH1 takes $O(\log m)$ time. Therefore, the total time is dominated by line 7 and is $O(n \log m)$, and each processor requires a constant number of memory locations. We can also achieve the same time complexity on a two-dimensional shuffle exchange computer with only $(mn)/(2 \log m)$ processors, as we have done for the hypercube.

6. OVERSIZED SVD PROBLEMS

Suppose there are vw processors in a hypercube SIMD computer. Consider the equivalent $v \times w$ rectangular array with cube connections. Without loss of generality, suppose $v \leq m$ and $w \leq n$; and m and n are multiples of v and w, respectively. We partition the m-by-n matrix A into h groups of w columns each: $C_1, C_2, \ldots, C_h$, as shown in Fig. 2 for $h = 3$. In this example, all column pairs which are orthogonalized in a sweep can be obtained by pairing the columns in the following way: (C_1, C_1), (C_1, C_2), (C_1, C_3), (C_2, C_2), (C_2, C_3). Therefore, the whole computation consists of processing all pairs of column groups by algorithm CCSVD. To process a pair of column groups, two matrix columns are stored in each column of processors, with m/v pairs of matrix elements stored in the local memory of each processor. $O(w(\log v + m/v))$ time is required to process a pair of column groups because each processor has to perform m/v multiplications and additions locally, and the summation of partial sums and the broadcast of parameters within

a column of processors each requires $O(\log v)$ time. The number of pairs of column groups is

$$h + (h-1) + (h-2) + \cdots + 2 + 1 = \frac{1}{2}h(h+1),$$

where $h = n/w$. Thus, the total time required to compute a sweep of SVD for an $m \times n$ matrix is $O\left(\frac{h}{2}(h+1)\, w(\log v + m/v)\right) = O(nh(\log v + m/v))$. For oversized SVD problems on shuffle-exchange computers, similar results can be obtained. Hence, we will not discuss them in detail in this paper.

7. CONCLUSIONS

Efficient SVD algorithms on hypercube and shuffle-exchange SIMD computers are presented in this paper. This paper has two main contributions. First, it improves the algorithms on hypercube computers in the literature [10]. Second, it proposes the concept of high-dimensional shuffle-exchange networks. Using this concept, we have successfully mapped SVD algorithms onto one-dimensional and two-dimensional shuffle-exchange computers and shown that their time complexities are on the same order as their hypercube counterparts, although a one-dimensional shuffle-exchange computer has only three connections and a two-dimensional shuffle-exchange computer only four connections within each PE.

REFERENCES

1. G.H. Golub and F.T. Luk, Singular value decomposition: Applications and computation, *Trans. 22nd Conf. Army Mathematicians*, ARO Report 77-1, 577–605 (1977).
2. G.H. Golub and C. Reisch, Singular value decomposition and least squares solutions, *Handbook for Automatic Computation, Vol. 2 (Linear Algebra)*, (Edited by J.H. Wilkinson and C. Reinsch), Springer-Verlag, New York, pp. 134–151, (1971).
3. G.H. Golub, W. Kahan and F.T. Luk, Computing the singular-value decomposition on the ILLIAC-IV, *ACM Trans. on Math. Software* **6**, 524–539 (1980).
4. R.P. Brent and F.T. Luk, The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays, *SIAM J. Sci. Statist. Computation* **6** (1), 69–84 (1985).
5. R.P. Brent, F.T. Luk and C.F. Van Loan, Computation of the singular value decomposition using mesh-connected processors, *J. VLSI Computer Systems* **1** (3), 243–270 (1985).
6. A.M. Finn, F.T. Luk and C. Pottle, Systolic array computation of the singular value decomposition, *Proc. SPIE Symp., Vol. 341, Real Time Signal Processing V*, 35–43 (1982).
7. D.E. Heller and I.C.F. Ispen, Systolic networks for orthogonal equivalence transformations and their applications, *Proc. 1982 Advanced Research in VLSI*, MIT, Cambridge, 113–122 (1982).
8. F.T. Luk, A triangular processor array for computing the singular value decomposition, TR84-625, Computer Science Dept., Cornell Univ. (1984).
9. R. Schreiber, A systolic architecture for singular value decomposition, *Proc. 1st Internat. Coll. on Vector and Parallel Comput. in Sci. Appl.*, Paris, 143–148 (1983).
10. H.Y.H. Chuang and L. Chen, Efficient computation of the singular value decomposition on cube connected SIMD machine, *Proc. of Supercomputing '89*, 276–282 (Nov., 1989).
11. M.R. Hestenes, Inverstion of matrices by biorthogonalization of related results, *J. Soc. Indust. Appl. Math.* **6**, 51–90 (1958).
12. P.J. Eberlein, On using the Jacobi method on the hypercube, *SIAM Proceedings of the Second Conference on Hypercube Multiprocessors*, 605–611 (1987).
13. C. Bischof, The two-sided Jacobi method on a hypercube, *SIAM Proceedings of the Second Conference on Hypercube Multiprocessors*, 612–618 (1987).
14. G.R. Gao and S.J. Thomas, An optimal parallel Jacobi-like solution method for the singular value decomposition, In *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 47–53, (1988).
15. H. Rutishauser, The Jacobi method for real symmetric matrices, In *Handbook for Automatic Computation, Vol. 2 (Linear Algebra)*, (Edited by J.H. Wilkinson and C. Reinsch), Springer-Verlag, Berlin, 202-211, (1971).
16. E. Dekel, D. Nassimi, and S. Sahni, Parallel matrix and graph algorithms, *SIAM Journal on Computing* **10** (4), 657–675 (1981).